AI Introduction Lecture

Josh Wilcox

September 30, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

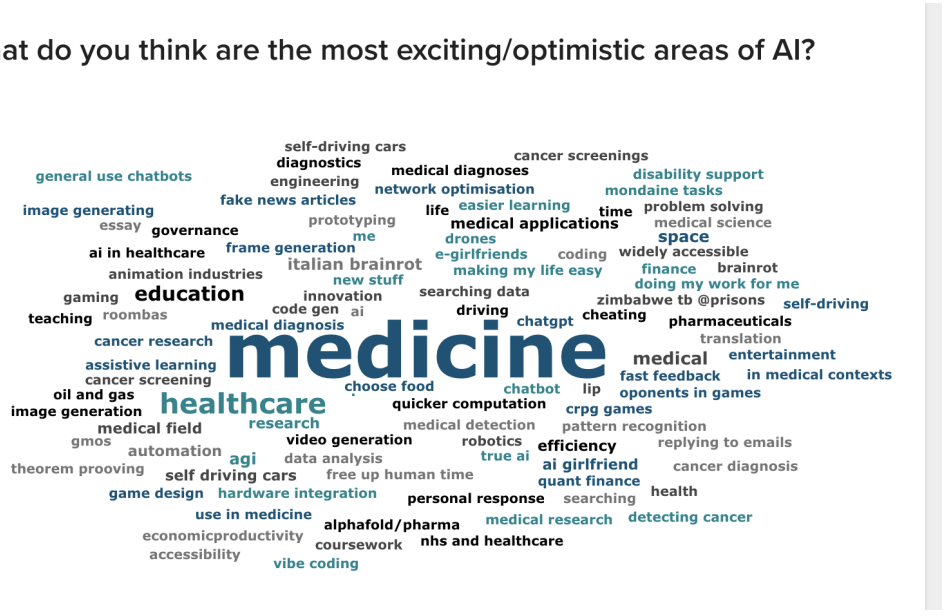
Table of Contents

Intro

Scope of AI Types

- Symbolic Reasoning and search (non learning)
  - Blind search, heuristic search, adversarial search
  - MCTS
  - constraint sat/solve
  - planning and logical reasoning
- Machine Learning
  - Non Adversarial Neural Networks
    - Regression models, decision trees, etc.
  - Artificial Neural Networks
    - Classifications, clustering, regression/curve fitting
    - Generative AI, CNNs, GANs, image generation, LLMs

AI Now and Future



Blind Search

Josh Wilcox

October 7, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Problem Solving

What is a problem?

- A problem - You are in some state of the world, it is not the state you want to be in.

How do you solve a problem?

- Series of actions to change the state of the world you are in to mean the problem is no longer present
- How do you know what actions to take?
  - Compare consequences of actions from a potential list and aim for the one that acheives the outcome you want
- You could also manage expectations of course

Classical AI approach to problem solving

- Assume a finite set of states
- Assume a set of actions is given
- Describe the new state of the world from each action
- If none of those states match the goal state
  - A sequence of problems may be required
  - How do we find that sequence

Single-State Problems Formulation

- Initial State
- Actions or Successor Functions
- Gloal Tests:
  - Explicit: x = ”At Bucharest”
  - Implicit: Checkmate(x)
- Cost of actions

Tree Searching

The Fringe (Frontier)

- **Fringe**: A data structure that stores nodes waiting to be expanded
- Also called the **frontier** or **open list**
- Contains leaf nodes of the search tree that haven't been explored yet
- Different search strategies use different fringe implementations:
  - **Queue (FIFO)**: Breadth-First Search
  - **Stack (LIFO)**: Depth-First Search
  - **Priority Queue**: Best-First Search, A*

General Tree Search Algorithm

- Initialize fringe with initial state
- **Loop:**
  - If fringe is empty, return failure
  - Remove node from fringe according to search strategy
  - If node contains goal state, return solution path
  - Expand node: generate all successor states
  - Add successor nodes to fringe

Key Components

- **Node**: Contains state, parent, action, path cost
- **Expansion**: Generate successors of a node
- **Search Strategy**: Determines which node to expand next
- **Solution**: Path from initial state to goal state

Important Considerations

- Tree search may revisit states (no cycle checking)
- Graph search maintains explored set to avoid cycles
- Search strategy determines:
  - Completeness (will it find a solution?)
  - Optimality (will it find the best solution?)
  - Time and space complexity

Search Strategies

What is a search strategy

- A search strategy is defined by **picking the order of node expansion**
- Strategies are evaluated along the following dimensions:
  - Completeness - Does it always find a solution if one exists
  - Time complexity - How many nodes generated
  - Space complexity - What is the max number of nodes
  - Optimality - Is it always least-cost

BFS

- Is complete?
  - Yes, there will always be a solution found, because all nodes will be visited at some point
  - Complete **iff** branching factor is finite
- Time Complexity?

$$1 + b + b^2 + \ldots + b^d = \mathcal{O}(b^{d+1})$$

  - Where d is depth **of the shallowest solution** and b is the max number of actions
- Space complexity
  - Every node kept in memory so $\mathcal{O}(b^{d+1})$
- Optimal **iff** all costs are equal

DFS

- Complete **iff** depth is finite
  - Always a solution found if all nodes can be visited and it doesn't get lost down one branch
- Optimal still is not guaranteed optimal even if all costs are equal - it can find really deep things before finding shallow things
- Time complexity

$$1 + b + b^2 + \ldots + b^m = \mathcal{O}(b^{m+1})$$

  - m is the **maximum depth of the tree** - which can be infinite
- Space Complexity

$$\mathcal{O}(bm)$$

Iterative Deepening Search

Depth Limited Search

- Same as DFS, but if you go in too deep - put the fries in the bag - and go back up
  - Nodes at max depth n simply do not expand

DLS Pseudocode

```
Node depthLimitedSearch(Node node, int limit) {
    if (goalTest(node.state)) {
        return node;
    }
    if (limit == 0) {
        return null; // cutoff
    }

    for (Node child : expand(node)) {
        Node result = depthLimitedSearch(child, limit - 1);
        if (result != null) {
            return result;
        }
    }
    return null; // failure
}
```

Iterative Deepening Search

- Iterate n from $0 \rightarrow n$, and keep running DLS on that up to a cutoff

```
for (int depth = 0; depth < infinity; depth++){
    result = DLS(problem, depth);
    if (!result.wasCutoff()){ return result }
}
```

How good is IDS

- **Optimality**: Yes, **iff** all costs are equal (finds shallowest solution)
- **Completeness**: Yes, **iff** branching factor is finite
- **Time Complexity**: $\mathcal{O}(b^d)$
- **Space Complexity**: $\mathcal{O}(bd)$

Time Complexity Explanation

- IDS performs multiple DLS calls with increasing depth limits
- At depth d, total nodes generated:

$$(d+1) \cdot 1 + d \cdot b + (d-1) \cdot b^2 + \ldots + 1 \cdot b^d$$

- This simplifies to approximately $\mathcal{O}(b^d)$
- The overhead is small because most work is done at the deepest level
- Each level i is searched $(d - i + 1)$ times, but the exponential growth dominates

Graph Search

What is Graph Search?

- Graph search avoids revisiting states by keeping track of **explored nodes**
- Uses an **explored set** (closed list) to remember visited states
- More memory efficient than tree search for problems with repeated states
- Prevents infinite loops in cyclic state spaces

Graph Search vs Tree Search

- **Tree Search**: May revisit same state multiple times
- **Graph Search**: Each state visited at most once
- Graph search trades memory for efficiency
- Better for problems where many paths lead to same state

Graph Search Algorithm

- Initialize fringe with initial state
- Initialize empty explored set
- **Loop:**
  - If fringe is empty, return failure
  - Remove node from fringe
  - If node contains goal state, return solution
  - Add node's state to explored set
  - Expand node, add new successors to fringe (if not already explored)

Bidirectional Search

How to do Bidirectional Search

- Do two searches - starting from initial and goal state
- This is done because $b^{d/2} * 2$ is much less than b^d

Heuristic Search

Josh Wilcox

October 7, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Best First Search

Idea of Best First Search

- Classical AI and ”Search” are synonyms
  - But classical AI is specifically about **reducing the amount of search you have to do**
- Best First Search uses an **evaluation function** for each node that **estimates its ”desirability”**
  - It expands the most desirable unexpanded node
- Implementation: Order nodes in fringe in decreasing order of **desirability**
  - For example, to find the shortest distance in a map, you could set the desirability to reference the straight-line-distance from each city to the destination

Greedy Best First Search

- Greedy BestFS does low amounts of search but is not always **optimal** - it sets the evaluation function directly to the heuristic function
- It expands a node that **appears** to be closest to the goal

How good is Greedy BestFS

- **Completeness** - No, can get stuck in loops if heuristic is bad
- **Time** - $\mathcal{O}(b^m)$ - good heuristics can give dramatic improvements
- **Space** - $\mathcal{O}(b^m)$ - keeps all nodes in memory
- Not optimal sets the evaluation function directly to the heuristic function

A* Search

Idea of A*

- Avoid expanding paths that are already expensive
- Evaluation function: $f(n) = g(n) + h(n)$
  - $g(n)$ - cost so far to reach n
  - $h(n)$ - estimated cost from n to goal
  - $f(n)$ - estimated total cost of path through n to goal

How A* Works - Step by Step

- **Start** with the initial node in the frontier (open list)
- **Repeat** until goal is found or frontier is empty:
  - Choose the node with **lowest f(n)** from frontier
  - If it's the goal, we're done!
  - Otherwise, expand it (find its neighbors)
  - For each neighbor:
    - Calculate $g(neighbor) = g(current) + cost(current, neighbor)$
    - Calculate $h(neighbor)$ using heuristic function
    - Calculate $f(neighbor) = g(neighbor) + h(neighbor)$
    - Add to frontier if not already explored
  - Move current node to explored list (closed list)

Admissible Heuristic

- A heuristic $h(n)$ is **admissible** iff for every node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost to reach the goal state from n
  - An admissible heuristic **never overestimates the cost to reach the goal** - therefore it is always **optimistic**
  - Straight line distance is an example of an admissible heuristic
- If $h(n)$ is admissible, A^* using `Tree-Search` is **optimal**
- You can relax a problem in order to acquire an admissable heuristic

---

Properties of A^*

- Complete - Yes, unless infinitely many nodes with $f \leq f(G)$
- Time - Exponential (in length of optimal solution)
- Space - Keeps all expanded nodes in memory
- Optimal - Yes - if admissible

Minimax Search

Josh Wilcox

October 19, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

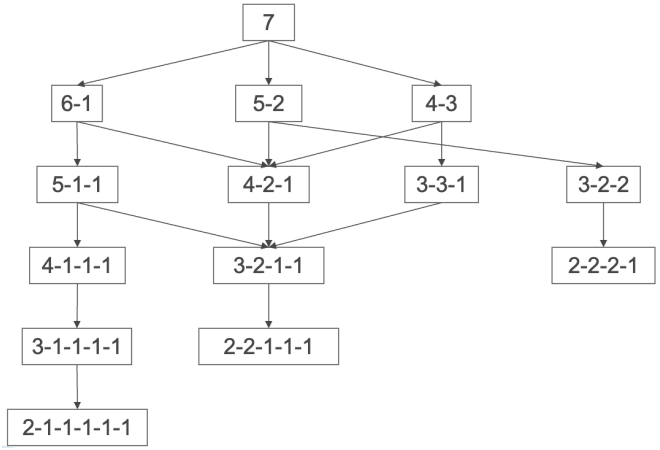
Your engagement and feedback help me a lot

Table of Contents

Adversarial Games

The Game of Nim

- Start with a single pile of seven matches, each player takes it in turn to take a pile of matches and split it into two differently sized piles of matches. The last player who is able to make a move is the winner
- Properties of the game of Nim:
  - Few legal states, and moves - and doesn't last long
  - Possible to construct a tree representing all the possible states of the game



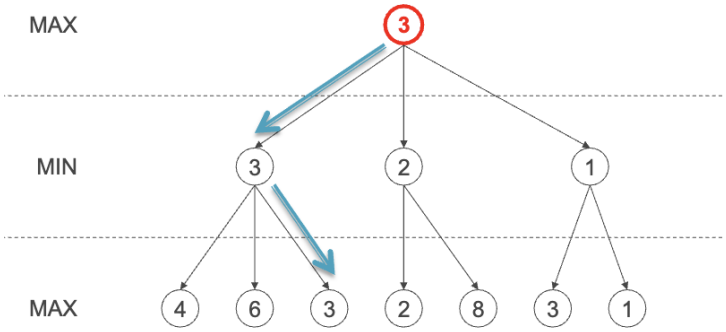
Minimax Search

- Minimax search is a view of search in adversarial games like the game of nim
- **MAX** is traditionally the computer player - and picks the moves whose heuristic value is the best
- **MIN** is the oppenent, and should pick moves that minimize MAX's advantage
- The classical AI makes the assumption that the oponent is playing well (playing the role of **MIN** correctly)

Minimax Process

Minimax Search Process

- Search to a given depth (through DLS) to build the game tree
- Evaluate heuristic for leaf nodes at the bottom of the tree
- **Bottom-up propagation**: Work backwards from leaf nodes to root
  - MAX nodes take the maximum of their child values (choosing best outcome)
  - MIN nodes take the minimum of their child values (opponent chooses worst outcome for us)
- This bottom-up approach ensures each node's value represents the best achievable outcome assuming optimal play from that position
- At the root, pick the action leading to the child with the best guaranteed score



Minimax Pseudocode

```
function MAXVALUE(state, depth){
    if (depth == 0) then return EVAL(state);
    v = -INF;
    for (s: SUCCESSORS(state)){
        v = MAX(v, MINVALUE(s, depth-1))
    }
    return v
}

function MINVALUE(state, depth){
    if (depth == 0) then return EVAL(state);
    v = -INF;
    for (s: SUCCESSORS(state)){
        v = MIN(v, MAXVALUE(s, depth-1))
    }
    return v
}
```

Evaluation Function

Evaluation Functions in Game AI

- **Purpose**: Estimate the utility/value of non-terminal game states when full search is impractical
- **Key Properties**:
  - Should correlate with actual game outcome probability
  - Computationally efficient (evaluated frequently)
  - Domain-specific features (material, position, mobility)
- **Common Approaches**:
  - Linear combination of weighted features
  - Piece values and positional advantages
  - Pattern recognition and tactical motifs
- **Output**: Numerical score representing estimated game value from current player's perspective

---

The horizon effect

- You can not exhaustively search most game trees
- Significant events may exist just beyond the maximum tree depth
  - For example, a capture of a queen may be just outside what we can see
- The further we can look ahead, the better our evaluation of a position
  - But clearly, this has time complexity issues

Quiescent Search

The need for Quiescent Search

- A bad evaluation function can lead to a wild swing in behaviour
  - For example, because of the horizon effect, it is very easy to arrive at a situation where a queen is taken one move after ours because of a limited search deptho
- Positions that can cause large swings in behaviour just one or two moves after are called **non-quiet** or **volatile**.

What Queiscent Search does

- Queiscent search makes it so you **never run the evaluation function on a volatile position**
- Instead of stopping **blindly** at the depth limit - a minimax algorithm first asks "is this position quiet":
  - If the position is quiet (no immediate threats) - it runs the evaluation function as normal
  - If the position is volatile (king in check, immediate threats) - it **continues searching** with **queiscent** search
    - The queiscent search only searches volatile moves until the dust settles

AlphaBeta

Josh Wilcox

October 14, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Going Beyond Minimax

Strong vs Weak Methods

- Strong methods take account of the structure of the whole game itself
- Weak methods can apply to any game
  - Minimax and Alpha-Beta are examples of weak methods
- We can limit the number of branches a classical AI makes by taking into account properties such as symmetry of the situation we are in
  - For example, multiple situations are directly equivalent in noughts and crosses, and we do not need to repeat a branch for these

Alpha-eta Pruning

- Alpha-beta pruning performs DFS but allows us to disregard certain branches of the tree
  - Alpha represents the lower bound on node value (the worst we can do)
    - Associated with MAX nodes
    - Never decreases
  - Beta represents the upper bound on node value (the best that we can do) on each branch
    - Associated with MIN nodes
    - Never increases

Alpha-Beta Process

- Initialize alpha = $-\infty$ and beta = $+\infty$
- At MAX nodes:
  - Update alpha = max(alpha, child value)
  - If alpha $\geq$ beta, prune remaining children (beta cutoff)
- At MIN nodes:
  - Update beta = min(beta, child value)
  - If beta $\leq$ alpha, prune remaining children (alpha cutoff)
- Pass alpha and beta values down to children
- Pruning occurs when alpha $\geq$ beta:
  - No need to explore remaining branches
  - The parent will never choose this path

Key Insight: If we find a move that's worse than what we already know the opponent can force, we can stop searching that branch.

Alpha-Beta vs Minimax

- Alpha-Beta is **guaranteed** to give the same values as minimax
- If the tree is ordered, complexity is $\mathcal{O}(b^{d/2})$
  - Minimax is $\mathcal{O}(b^d)$

Local Search

Josh Wilcox

October 14, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Local Search

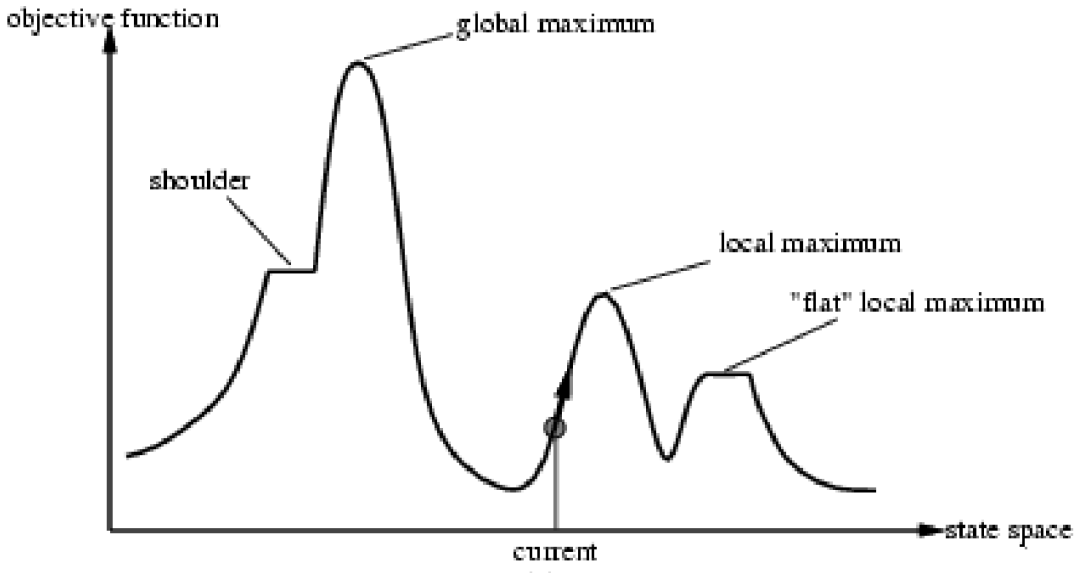
When is it used

- When the path to the solution is irrelevant, only the final solution matters
- When the state space consists of complete configurations that can be evaluated
- When systematic search is impractical due to large search spaces
- Examples: N-Queens, Traveling Salesman Problem, scheduling problems

---

Hill-Climbing Search

- **What is Hill-Climbing?**
  - Like climbing a hill in fog - you can only see your immediate surroundings
  - Always move to the neighbor that looks better than your current position
  - Stop when no neighbor is better (you've reached a "peak")
- **How it works:**
  - Start with a random solution
  - Look at all neighboring solutions (small changes to current solution)
  - Move to the best neighbor if it's better than current
  - Repeat until no improvement possible
- **The Problem: Local Maxima**
  - You might climb the wrong hill and get stuck on a small peak
  - Can't see the bigger, better hill nearby due to the "fog"
  - Example: In N-Queens, might find a configuration with few conflicts but miss the perfect solution



- **Solutions to Getting Stuck:**
  - **Random Restart:** Start climbing from different random points
  - **Sideways Moves:** Sometimes move to equally good neighbors
  - **Simulated Annealing:** Occasionally make "bad" moves to escape local peaks

Simulated Annealing

What is Simulated Annealing

- Named after the metallurgical process of slowly cooling metal to remove defects
- Like hill-climbing, but occasionally allows ”downhill” moves to escape local peaks
- Think of it as having a ”shake” that gets gentler over time

How it improves Hill-Climbing

- Hill-climbing gets stuck on local peaks - can’t go down to find better hills
- Simulated annealing starts ”hot” - willing to make many bad moves to explore
- As it ”cools down,” it becomes more selective, like hill-climbing
- Eventually becomes pure hill-climbing when fully ”cooled”

Simulated Annealing Temperature Mechanism

- **High temperature (T):** Accept most moves, even bad ones (lots of exploration)
- **Low temperature (T):** Accept mainly good moves (focused search)
- Temperature decreases over time according to a cooling schedule
- If cooled slowly enough, guaranteed to find global optimum

Decision Rule

- Always accept better moves, accept worse moves with probability $e^{-\Delta E/T}$
- ΔE = how much worse the move is
- Higher T = more likely to accept bad moves
- Lower T = less likely to accept bad moves

Local Beam Search

What is Local Beam Search?

- Instead of climbing one hill at a time, you have k search parties climbing different hills simultaneously
- Like having multiple flashlights in the fog instead of just one
- All search parties communicate and share information about what they find

How it works

- Start with k randomly placed search parties (states)
- Each party explores their immediate surroundings (generates successors)
- All parties report back their findings to a central command
- Keep only the k best locations found across all parties
- Redistribute parties to these best locations and repeat

Key Advantages

- **Information Sharing:** If one party finds a promising area, other parties can join them
- **Parallel Exploration:** Multiple areas searched simultaneously
- **Dynamic Focusing:** Search effort automatically concentrates on most promising regions
- Example: If 3 parties find mediocre spots but 1 party finds an excellent area, all parties might move to explore that excellent area

Genetic Algorithms

What are Genetic Algorithms?

- Inspired by biological evolution - ”survival of the fittest”
- Instead of one search party, you have a whole population exploring simultaneously
- Like breeding programs where you combine the best traits from successful parents
- Think: if two chess players each have good strategies, their ”offspring” might have even better combined strategies

How it works (8-Queens Example)

- **Population:** Start with many random queen arrangements (e.g., 100 different board configurations)
- **Fitness:** Measure how good each arrangement is (fewer queen conflicts = higher fitness)
- **Selection:** Pick the best arrangements as ”parents” (those with fewer conflicts)
- **Crossover:** Combine two parent arrangements to create ”children”
  - Take first 4 columns from Parent A, last 4 from Parent B
- **Mutation:** Occasionally randomly move a queen to add variety
- Repeat for many generations until perfect solution emerges

Why does this work?

- Good partial solutions (like ”no conflicts in these 3 columns”) get passed down
- Bad arrangements gradually disappear from the population
- Crossover can combine good features from different solutions
- Mutation prevents getting stuck by introducing new possibilities
- Over time, the population evolves toward better and better solutions

Constraint Specification Problems

Josh Wilcox

October 21, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

What are CSPs

Standard Search Problems

- Normally, in a standard search problem, a **state** is a "black box"
  - A black box supports successor functions, heuristic functions, and goal tests

How CSPs are Different

- A state is defined by variables X - with values from domain D
- Goal tests are a **set of constraints** that specify allowable combinations of values for subsets of variables
- Allows for general-purpose algorithms with more power than standard search algorithms

Constraint Graphhs

- A constraint graph contains nodes for variables, and edges for constraints

Searching CSPs

Standard Search Formulation

- Set **initial state** - to the empty assignment {}
- The **successor function** assigns a value to an unassigned variable that does not conflict with the current assignment
  - Fail (stop going) if no legal assignments
- The **goal test** is when the assignment is complete

Backtracking

- In a CSP, we want to go as deep as we can, as going to the bottom possible level of a search tree for this kind of issue **will be a solution**
- So to solve a CSP we can employ DFS of depth n where n is the number of unassigned variables
- **Backtracking and DFS are not exactly the same though!**
  - Backtracking search just applies **one action** to produce one successor node and continues own down - not all expansions occur on each.
    - So it feels more depthy thanDFS

Ordering Search

- Pick most **constrained** unassigned variable first
- Failing that (draw or not applicable) pick most **constraining** unassigned variable

Planning

Josh Wilcox

October 21, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Planning in AI

- We want to be able to estimate how many steps there are from the start to the goal to minimize

---

Evaluating Real world problem difficulty

- Which actions are relevant
  - Exhaustive vs Backward Search for example
- What are good heuristic functions
  - Do we have a good estimate of the cost of the state
- How to decompose a problem
  - Most real-world problem are *nearly* fully decomposable

Planning Lanugage

- We need a more sophisticated way of describing problems
- In a planning problem, actions are defined in a way that it is specific to what precise aspects of the world it will change
  - More of an update than a full redesign of the whole world

---

Language Features

- Representation of states:
  - Decompose the world in **logical** conditions and represent a state as **a conjunction of positive literals**
- Representation of goals
  - Conjunction of positive ground literals
- Representation of Actions
  - Precondition + Effect (both conjunctions of function-free literals)
-

Progression and Regression

Progression Planners

- Progression planners do **forward state-space search**
- They consider the effect of all possible actions in a given state

Regression Planners

- Regression planners do **backward state-space search**
  - Start from goal and try and reach
- To acheive a goal, what must have been true in the previous state

Partial-Order Plans

- Progression and regression planning are *totally ordered plan search*
  - They can not take advantage of problem decomposition
  - Decisions must be made on how to sequence actions on all the subproblems
- Least commitment strategy
  - Delay choices **that do not need to be made yet** during construction on the plan
- A partial-order plan can place two actions into a plan **without fixing which comes first**

History of AI

Josh Wilcox

October 23, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

History of AI

- When clockwork was the most powerful technology around, it was used as amodel for how intelligence works
  - Then things moved from clockwork → automata → computers *to* artificial neural networks → LLMs
- Makes you question are LLMs just as mindless as the previous technologies from before

What is Intelligence

Josh Wilcox

October 28, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Minds

Mind Over Matter

- A solution to finding what the mind is to consider the mind to be **immaterial**
- Mind is more like "soul" or "self" over a body part like the foot
- If the mind is immaterial, how does it interact with the body which **is** material
- Materialists beleive that **brains cause minds** - but the question is how

The problem of other minds

- We can be certain as people that **we alone definitely have minds**
- But how can we tell if other things also have minds
  - Do we look inside?
  - What are we actually looking for? Neurons?
  - How are neurons any different to complex machinery?

Turing Tests

Behaviour and Turing Tests

- Instead of focussing on minds, some people just care about **behaviour**
- Turing focussed on behaviour in the **Turing Test**
  - If A is a computer, B is a human, and C can not tell which of A and B is the computer imitating the other, then A must be intelligent
- Benefits of the Turing test:
  - Avoids difficult wishy-woshy psychology questions like "what is conciousness"
  - Quantifiable result, can be implemented
- But the turing test comes with its own problems:
  - What do you even ask?
  - What are you looking for in an answer?
  - Doesn't factor in task-based intelligence
  - Is knowledge intelligence?
  - Is mimicing emotion the same as having emotion

Symbol-Grounding Problem

- Are just using symbols to signify thoughts enough to prove intelligence?
- Classical AI takes thoughts to be like sentences:
  - Cat is on the mat = `On(Cat,Mat)`
- Thought is manipulation of the expressions like above
  - Godel places formal limits on this system
  - Wittgensetein argued that some complex concepts are not easily captured by prepositions in this way

Searle's Chinese Room

The problem of the chinese room

- John Searle's chinese room argument uses **symbol grounding** to challenge AI
- He imagines being part of a room that can pass the turing test - but in **Chinese** (language he can't understand)
- He's just following a rulebook for manipulating Chinese symbols. He doesn't understand Chinese, but from the outside, his responses look perfectly fluent.
- The room's **syntax** captures **rules of content** - but the room doesn't have intrinsic meaning to him
- Computers will only follow synctactic rules, not semantic meanings

Grounding Symbols

- There are two possibilites for the meaning of grounding symbols:
  - An interpreter or observer does the symbols
  - The system does the symbols (where the system knows what it means itself)
- Most AI systems do option 1 - but true AI must take option 2

Sub-Symbolic AI

- Artificial **Neural Networks** do not require **explicit symbols**
  - The things that are being transformed are vector fields instead of just being given commands one at a time
- LLMs do do tokenization
  - Turns them into symbols then uses a neural network on it

The Frame Problem

The Frame problem

- The **Frame Problem** concerns how an intelligent agent can determine which facts about the world remain **unchanged** after an action occurs.
- In logic-based AI and planning systems, actions are usually defined by their effects. However, it is computationally expensive and conceptually difficult to explicitly state every fact that *does not* change.
- For example, if an agent performs the action `Fly(Plane1, Airport1, Airport2)`, we must record that the plane is no longer at Airport1 and is now at Airport2.
- The challenge is how to represent that all other properties of the plane (its color, weight, etc.) and unrelated facts about the world *remain the same*—without explicitly listing them.
- In the **narrow sense**, the frame problem is about specifying non-changing conditions efficiently in logical representations.
- In the **broader philosophical sense**, it raises the question of how minds or machines can focus on what is *relevant* to an action or event without being overwhelmed by irrelevant details.
- For example, when moving a cup of tea, we intuitively know that the temperature of the tea or the number of chairs in the room do not change—yet formal systems struggle to encode this kind of common-sense stability.

What is Cognition

Josh Wilcox

October 30, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Cognition

Cognition vs Intelligence

Propositional Logic

Josh Wilcox

November 4, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Logical and Knowledge Based Agents

Logical Agents

- We want to design AI agents that make **decisions** about the **environment** they are in
- Agents should be able to infer new information and new knowledge and dynamically **add it to what they already know**
- A **knwledge base** is a set of sentences that describs facts about the world
  - Each sentence is expressed in a language called **knowledge representation language** - they all represent some assertion about the world
- Inference must obey the requirement that when one **queries the knowledge base**, the answer should follow from what has been told

Example of a knowledge-based Agent

function KB-AGENT(*percept*) **returns** an *action*
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE($percept$, t))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action$, t))
  $t \leftarrow t + 1$
  return *action*

- Given a percept, the agent adds the percept to the knowledge base
- It then asks the KB for the best action
- And then tells the KB that action has indeed been taken
- For logical reasoning, given that information is correct - the conclusion is guaranteed to be correct as well

Propositonal Logic

Connectives

- We can do all we want for propositional logic using:
  - Imply: $\implies$
  - Conjunction: $\wedge$
  - Negation $\neg$
- THe other connectives can be defined from this:
  - $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$
  - $\varphi \iff \psi = (\varphi \implies \psi) \cap (\psi \implies \varphi)$

Valuations

- **Negations:**
  - $v(\neg\varphi) = 1$ if $v(\varphi) = 0$
  - 0 otherwise
  - $\neg\varphi$ is true IFF φ is false.
- **Conjunctions:**
  - $v(\varphi \wedge \psi) = 1$ if $v(\varphi) = v(\psi) = 1$
  - 0 otherwise
  - $\varphi \wedge \psi$ is true IFF both φ and ψ are true.
- **Disjunctions:**
  - $v(\varphi \vee \psi) = 0$ if $v(\varphi) = v(\psi) = 0$
  - 1 otherwise
  - $\varphi \vee \psi$ is true IFF at least one formula in $\{\varphi, \psi\}$ is true.
- **Implications:**
  - $v(\varphi \implies \psi) = 0$ if $v(\varphi) = 1$, $v(\psi) = 0$
  - 1 otherwise
  - $\varphi \implies \psi$ is true IFF either φ is false or both φ and ψ are true.
- **Double Implications:**
  - $v(\varphi \iff \psi) = 1$ if $v(\varphi) = v(\psi)$
  - 0 otherwise
  - $\varphi \iff \psi$ is true IFF φ and ψ have the same truth-value

Proofs

Semantic Consequence

- Given a knowledge base KB, we denote by $Mod(KB)$ the **set** of **valuations** that make all the formulas in the KB true
- Given a KB, and formula φ, we say φ is a **semantic consequence** of KB ($KB \vDash \varphi$) if every valuation that makes the formulas in KB true also makes φ true
- This means that:

$$Mod(KB) \subseteq Mod(\varphi) \text{ if } KB \vDash \varphi$$

Propositional Logic Axioms

$$\varphi \implies (\psi \implies \varphi)$$
$$(\varphi \implies (\psi \implies \gamma)) \implies ((\varphi \implies \psi) \implies (\varphi \implies \gamma))$$
$$(\neg\varphi \implies \neg\psi) \implies (\psi \implies \varphi)$$
$$\text{With } \varphi \text{ and } \varphi \implies \psi \text{ infer } \psi$$

Proofs in Propositional Logic

- A knowledge base (KB) is a set of propositional formulas.
- Given a KB, a proof is a finite sequence of formulas $\phi_1, \ldots, \phi_n$ where each ϕ_i is
  - a formula from KB, or
  - an instance of an axiom (A1-A3), or
  - derived from ϕ_j, ϕ_k, with $j, k < i$ by using modus ponens.
- Given a KB and a formula ϕ, we say that ϕ is a logical consequence of KB, denoted $KB \vdash \phi$ if there exists a proof of ϕ from KB.
- A formula ϕ is called a theorem if it can be derived from the axioms by using modus ponens, i.e. $\vdash \phi$.

Soundness and Completeness

Soundness

- $\vdash$ is about **deductive inference and proof** - *syntactical*
- $\models$ is about **relationship** between models - *semantical*
- Propositional logic is **sound** meaning:

$$\text{If } KB \vdash \varphi \text{ then } KB \models \varphi$$

Completeness

- Propositional logic is also complete:

$$\text{If } KB \models \varphi \text{ then } KB \vdash \varphi$$

- Semantic and syntactic consequences are equivalent in classical propositional logic
- every theorem is a tautology, and every tautology is a theorem

Inference from Knowledge Baase

Inference

- We start with semantic inference, we want to check that for a KB and formula φ, it is the case that:

$$KB \vDash \varphi$$

- The simplest algorithm enumerates all models of KB and φ and checks if:

$$Mod(KB) \subseteq Mod(\varphi)$$

Proofs

- We can use BFS, DFS, etc. to apply propositional logic inference rules
- We just need to define the proof problem as follows:
  - `INITIAL STATE`: The initial knowledge base
  - `ACTIONS`: All the inference rules applies to all the formulas that match the ”From” part of the inference rule
  - `RESULT`: The result of an action is to add the formula from the ”infer” part of the inference rool
  - `GOAL` : The state that contains the formula we are trying to prove
- Finding a proof in this way can be more efficient because the proof can ignore irrelevant propositions - no matter how many of them there are.

Resolution Algorithm

Josh Wilcox

November 6, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Resolution

What is Resolution

- We can create algorithms for **finding proofs** based on a single inference rule called **resolution**
- Recall that a search algorithm is **complete** if it is guaranteed to find a solution
- When a **complete** search algorithm is paired with the resolution rule, we obtain a **complete inference algorithm**
- This means the search algorithm with a resolution rule will decide whether:

$$KB \vDash \varphi$$

Unit Resolution

- Unit resolution is a **special case** of general resolution where one clause is a single literal:

$$\frac{l_1 \vee \ldots \vee l_k, \quad m}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k}$$

  - Here, m is a single literal (unit clause) and l_i and m are **complementary**
  - The complementary literal l_i is removed from the first clause
- **In simple terms:** If you know a fact is definitely true (like "it's raining"), and you have another statement that includes the opposite of that fact (like "it's NOT raining OR take an umbrella"), you can simplify it to just "take an umbrella"
- **Example:** From "A" and "(NOT A OR B OR C)", we can derive "(B OR C)"
- A single literal can be viewed as a disjunction of one literal, also known as a **unit clause**

General Resolution Rule

- The general resolution rule combines two clauses that contain complementary literals:

$$\frac{l_1 \vee \ldots \vee l_i \vee \ldots \vee l_k, \quad m_1 \vee \ldots \vee \neg l_i \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{n-1}}$$

  - The literals l_i and $\neg l_i$ are **complementary** (one negates the other)
  - These complementary literals **cancel out**, and the remaining literals form the new clause
- **In simple terms:** If you have two statements connected by "OR", and they contain opposite claims (like "it's raining" and "it's NOT raining"), you can combine them by removing the contradictory parts and keeping everything else
- **Example:** From "(A OR B)" and "(NOT A OR C)", we can derive "(B OR C)"
- For instance:

$$\frac{(P_{1,1} \vee P_{3,1}), (\neg P_{1,1} \vee \neg P_{2,2})}{P_{3,1} \vee \neg P_{2,2}}$$

  - The complimentary rules ($P_{1,1}$ and $\neg P_{1,1}$) have been removed through resolution
- The removal of multiple copies of literals is called **factoring**

Conjuctive Normal Form

What is CNF

- Conjunctive Normal Form is a form of predicates that consist of **conjunctions** of **dysjunctive clauses**

$$(a \vee b \vee c) \wedge (d \vee e) \wedge f$$

- The resolution rule **only applies to these clauses**

Converting to CNF

- We can convert to CNF using the definitions from predicate logic and de morgans laws
- For example,

Convert $B_{1,1} \iff (P_{1,2} \vee P_{2,1})$ to CNF

$$\begin{aligned}
B_{1,1} \iff (P_{1,2} \vee P_{2,1}) &\equiv (B_{1,1} \implies (P_{1,2} \vee P_{2,1})) \wedge (P_{1,2} \vee P_{2,1}) \implies B_{1,1} && \text{iff definition} \\
&\equiv (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) && \text{implies definition} \\
&\equiv (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) && \text{de morgans} \\
&\equiv (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (B_{1,1} \vee \neg P_{1,2}) \wedge (B_{1,1} \vee \neg P_{2,1}) && \text{distributivity}
\end{aligned}$$

Using the Resolution Algorithm

Deduction Theorem

- The deduction theorem says that:

$$\psi \vDash \varphi \quad \text{IFF} \quad \vDash (\psi \implies \varphi)$$

- The deduction theorem tells us that φ is a logical consequence of ψ if and only if $\psi \implies \varphi$ **is a tautology**
- An equivalent formulation is as follows:

$$\psi \vDash \varphi \quad \text{IFF} \quad \vDash \neg(\psi \wedge \neg\varphi)$$

- In this case, φ is a logical consequence of ψ IFF $\psi \wedge \neg\varphi$ is a contradiction

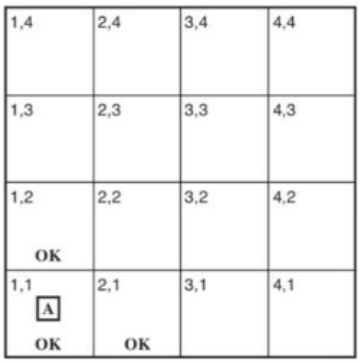
How the resolution works

- The resolution algorithm uses this deduction theorem to show that:

$$KB \vDash \varphi$$

- It does this by showing that $KB \wedge \neg\varphi$ is a contradiction and **unsatisfiable**
- It starts therefore by converting $(KB \wedge \neg\varphi)$ into a CNF
- The resolution rule is then applies to the resulting clauses
- Each pair that **contains complementary literals** are resolved to produce a new clause - which is added to the set if not already present
- This process continues until either of the following happen:
  - No more clauses can be added - in which case KB **does not entail** φ
  - Two clauses resolve to yield the empty clause: In which case KB **does entail** φ

Wumpus Example



We can apply the resolution procedure to a very simple inference in the wumpus world.

When the agent is in (1, 1), there is no breeze, so there can be no pits in neighbouring squares.

We have

$$KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

and we want to prove that $\neg P_{1,2}$.

- We want to prove:

$$(B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \wedge \neg B_{1,1}) \models \neg P_{1,2}$$

- So we do this by proving $(B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \wedge \neg B_{1,1}) \wedge \neg\neg P_{1,2}$ is a contradiction

Wumpus Example - Steps 1 to 4

Step 1: Write out what we know and what we want to prove is false

- **What we know (our knowledge base):**
  - $B_{1,1} \iff (P_{1,2} \vee P_{2,1})$ - "There's a breeze at position (1,1) if and only if there's a pit at (1,2) or (2,1)"
  - $\neg B_{1,1}$ - "There is NO breeze at position (1,1)"
- **What we want to prove:** $\neg P_{1,2}$ - "There is NO pit at position (1,2)"
- **By the deduction theorem:** To prove this, we assume the opposite is true (that there IS a pit at (1,2)), and show this leads to a contradiction
- So we need to show: $(B_{1,1} \iff (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1} \wedge P_{1,2}$ is unsatisfiable

---

Step 2: Convert $B_{1,1} \iff (P_{1,2} \vee P_{2,1})$ to CNF

- We already did this earlier! Recall:

$$\begin{aligned} B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \equiv & (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \\ & \wedge (B_{1,1} \vee \neg P_{1,2}) \\ & \wedge (B_{1,1} \vee \neg P_{2,1}) \end{aligned}$$

Step 3: Write out all our clauses in CNF

- Now we have all the pieces of our knowledge base in CNF, plus the negation of what we want to prove:
  - $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$ (from the biconditional)
  - $B_{1,1} \vee \neg P_{1,2}$ (from the biconditional)
  - $B_{1,1} \vee \neg P_{2,1}$ (from the biconditional)
  - $\neg B_{1,1}$ (fact: no breeze)
  - $P_{1,2}$ (assumption: pit at (1,2) - trying to contradict)
- **Goal:** Apply resolution repeatedly until we derive the empty clause ($\square$)

Step 4: First resolution - Combine clauses (2) and (4)

- $B_{1,1} \vee \neg P_{1,2}$ and $\neg B_{1,1}$
- The complementary literals $B_{1,1}$ and $\neg B_{1,1}$ cancel out
- **Result:** $\neg P_{1,2}$ (clause 6)
- **Meaning:** "There is NO pit at (1,2)"

Wumpus Example - Steps 5 & 6

Step 5: Second resolution - Combine clauses (5) and (6)

- $P_{1,2}$ (our assumption) and $\neg P_{1,2}$ (what we just derived)
- These are **complementary literals** - they directly contradict each other!
- When we apply resolution, both literals cancel out
- **Result:** $\square$ (the empty clause)

Step 6: Conclusion

- **What does this mean?**
  - The empty clause ($\square$) represents a contradiction
  - We've shown that assuming $P_{1,2}$ (pit at position (1,2)) leads to a logical impossibility
  - Therefore, our assumption must be FALSE
  - This proves that $\neg P_{1,2}$ is true - there is NO pit at position (1,2)!
- **Final result:** $(B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \wedge \neg B_{1,1}) \models \neg P_{1,2}$ ✓

First Order Logic

Josh Wilcox

November 6, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

First Order Logic

What FOL Models

- Objects - things with individual identities
- Properties - distinguish objects from others
- Relations - relations that hold among sets of objects
- Functions - subset of relations where there is only one ”value” for any given input
- Quantifiers:
  - Universal ($\forall x$)
  - Existential ($\exists x$)

What FOL Provides

- **Constant Symbols** - representing individuals
  - Mary, 3, Green
- **Function Symbols** - Map individuals to individuals
  - `father-of(Mary) = John`
  - `color-of(Grass) = Green`
- **Predicates** - Maps individuals to **truth values**
  - `greater(5,3)`
  - `green(Grass)`
  - `color(Grass, Green)`

Sentences

- A **term** is a constant symbol, a variable symbol or an **n-place function of terms**
  - So $f(x_1, \ldots, x_n)$ is an example of **a** term
- An **atomic sentence** is an n-place **predicate** of n terms
- A **complex sentence** is formed from atomic sentences connected by the logical connectives
- A **quantified** sentence is one that has quantifiers $\forall$ and $\exists$
- A **well-formed formula** is a sentece where every involved variable is **quantified**

Probabilistic Reasoning

Josh Wilcox

November 11, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Probability Theory

Axioms and Rules

- Probability has a value between 0 and 1
$$0 \leq P(A) \leq 1$$
  - A is an *event* belonging to probability space Ω
- Summing over all probabilities of all events in Ω gives 1
- $P(TRUE) = 1, P(FALSE) = 0$
- AND and OR statements are links:
$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$
- Conditional Probability Works as Follows:
$$P(B \mid A) = \frac{P(A \cap B)}{P(A)}$$
$$P(A \cap B) = P(B \mid A)P(A)$$
- Law of Total Probability:
$$\begin{aligned} P(B) &= \sum_a P(B \cap (A = a)) \\ &= \sum_a P(B \mid A = a)P(A = a) \end{aligned}$$

Bayes Theorem

$$\begin{aligned} P(A \cap B) &= P(B \mid A)P(A) \\ &= P(A \mid B)P(B) && \text{Chain Rule twice} \\ &\Downarrow \\ P(B \mid A)P(A) &= P(A \mid B)P(B) \\ &\Downarrow \\ P(B \mid A) &= \frac{P(B)P(A \mid B)}{P(A)} \\ &= \frac{P(B)P(A \mid B)}{\sum_b P(A \cap (B = b))} && \text{Law of total probability} \\ &= \frac{P(B)P(A \mid B)}{\sum_b P((B = b) \mid A)P(B = b)} \end{aligned}$$

Probabilistic Inference

Knowledge Representation

- We use probabilities to capture uncertainty in our knowledge
- This is done with **simple correlations** between probabilities
  - What can we infer given two events happeningv
- **Inference**: Derive extra *conclusions* from **observed data**
  - Can include much more complicated networks of correlations
  - Can use Baye's theorem using **Bayesian Networks**

Inference with Joint Distributions

- The simplest way to do inference is to look at **joint distribution** of probability variables
  - This is especially useful when the set of binary variables is small
- Uses **truth tables**

Male	Long hours	Rich	Probability
T	T	T	0.13
T	T	F	0.11
T	F	T	0.10
T	F	F	0.34
F	T	T	0.01
F	T	F	0.04
F	F	T	0.02
F	F	F	0.25

Q: P(L | M) = ?

A: P (L|M) = P(L and M)/P(M) = (0.13 + 0.11)/(0.13+0.11+0.10+0.34) = 0.35

- Joint distributions captures all interconnections and dependencies
- Joint distributions **do not scale well in practice** - it is a brute force solution

Conditional Law of Total Probability

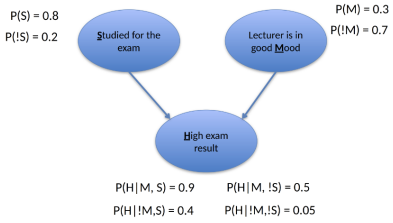
$$P(A \mid B) = \sum_a P(A \mid B, C_a)\, P(C_a \mid B)$$

- $\{C_a\}$ is a partition (mutually exclusive and exhaustive) of the relevant space.
- This marginalises over the conditional cases C_a to compute $P(A \mid B)$.
- Useful when $P(A \mid B, C_a)$ and $P(C_a \mid B)$ are easier to obtain than $P(A \mid B)$ directly.
- Ensure normalization: $\sum_a P(C_a \mid B) = 1$.

Bayesian Networks

What are Bayesian Networks

- They use **graphical representation** to capture dependencies between random variables



- Captures only the necessary relationships
  - Studying for exams and lecturers being in good mood does not need to be expressed in this context

Example: Computing $P(M|H)$ Step-by-Step

- **Apply Bayes' Theorem:**

$$P(M|H) = \frac{P(M)P(H|M)}{P(H)}$$

- **Identify what we need:**
  - $P(M)$: Prior probability (given directly)
  - $P(H|M)$: Likelihood (needs computation)
  - $P(H)$: Evidence (needs computation)
- **Compute $P(H|M)$ using Conditional Law of Total Probability:**
  - Marginalize over S:

$$P(H|M) = P(H|M,S)P(S|M) + P(H|M,\neg S)P(\neg S|M)$$

  - Since M and S are independent: $P(S|M) = P(S)$

$$P(H|M) = P(H|M,S)P(S) + P(H|M,\neg S)P(\neg S)$$

- **Compute $P(H)$ using Law of Total Probability:**
  - Marginalize over both M and S:

$$\begin{aligned} P(H) = {} & P(H|M,S)P(M \cap S) + P(H|M,\neg S)P(M \cap \neg S) \\ & + P(H|\neg M,S)P(\neg M \cap S) + P(H|\neg M,\neg S)P(\neg M \cap \neg S) \end{aligned}$$

  - Exploit independence of M and S:

$$\begin{aligned} P(H) = {} & P(H|M,S)P(M)P(S) + P(H|M,\neg S)P(M)P(\neg S) \\ & + P(H|\neg M,S)P(\neg M)P(S) + P(H|\neg M,\neg S)P(\neg M)P(\neg S) \end{aligned}$$

- **Substitute back into Bayes' Theorem:**

$$P(M|H) = \frac{P(M) \cdot P(H|M)}{P(H)}$$

where $P(H|M)$ and $P(H)$ are computed from steps 3 and 4.

Properties of Bayesian Networks

Properties

- Must be **Directed Acyclic Graphs**
- Memory efficient - but do not economise on computing times
- Easy for humans to interpret and explain

Advantages of Bayes Theorem

- Captures uncertainty of knowledge in an elegant and simple way
- Allows us to integrate prior knowledge into the reasoning process
- We can always update our beleif about the world using Bayes Theorem

Types of Learning

Josh Wilcox

November 20, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Machine Learning

- **Actions vs Behaviours**: Behaviours involve **context** that surround actions

---

What is Machine Learning

- Field that gives a computer the **ability to learn** without being **explicitly programmed**
- A program should not be explicitly told exactly what to do next in the each of the steps of computation.
- An ML program **learns** from experience E with respect to some class of tasks T and performance metric P, if its performance of tasks in **T**, as measured by P improves with experience E

Supervised Learning

- It contains a **vector** with mappings:

$$\text{Training Data: } \{(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)\}$$

  - y_1 is the **label** for the **feature vector** x_1
- Labels can be **discrete values**: e.g. { spam, nospam } or { cancer, healthy } or **continuous values** - such as stock market tickers etc.
- **Example of Training Data for Supervised Learning:**

day	outlook	temp	humid	wind	tennis'
1.	sun	hot	high	weak	no
2.	sun	hot	high	strg	no
3.	cloud	hot	high	weak	yes
4.	rain	mild	high	weak	yes
5.	rain	cool	norm	weak	yes
6.	rain	cool	norm	strg	no
7.	cloud	cool	norm	strg	yes
8.	sun	mild	high	weak	no
9.	sun	cool	norm	weak	yes
10.	rain	mild	norm	weak	yes
11.	sun	mild	norm	strg	yes
12.	cloud	mild	high	strg	yes
13.	cloud	hot	norm	weak	yes
14.	rain	mild	high	strg	no

- What a feature vector could look like:

$$x_1 = \begin{bmatrix} sun \\ hot \\ high \\ weak \end{bmatrix}$$

- Classifiers find a **function** form a feature vector (like above) to a label

---

Unsupervised Learning

- Input data: $\{x_1, x_2, \dots, x_n$
- **NO LABEL INFORMATION** - only involves feature vectors
- It **groups** data that it sees are close to each other
- Useful in:
  - Anomoly detection, knowledge discovery, etc.

Example Problems

Handwriting

- T: Recognising and classifying handwritten words
- P: Percent of words correctly classified
- E: Database of handwritten words with given classification

Self-Driving Cars

- T: Driving on a public four-lane motorway using vision sensors
- P: Average distance travelled before an error (judged by a supervisor)
- E: Sequence of images and steering commands recorded while observing a human driver

Decision Trees

Decision Trees

- DT learning is a method for approximating discrete-valued target functions, in which the **learned** function is a DT
- $h(x) \rightarrow y$ **takes the form of a tree**
- Each node represents a **test on a feature** - each leaf is a **final decision** (classification)
- Ordering of the features we test matters
- Any function in DNF can be expressed as a decision tree:
  - Each conjunction is a path

Formal Information Gain

- Information gain can be **mathematically defined** in terms of **entropy**
- It is the reduction of **entropy** observed when selecting a specific feature from any tree

---

Building a DT

- Always test the **most important attribute first**, then recursively solve the smaller subproblems in a divide-and-conquer manner
  - Most important attribute - the one that makes the most difference to classification

Example Decision Tree

- Generate a decision tree with minimum number of nodes based on the following table.
- Use only the decision process we covered in the lecture

Author	Length	Thread	User Action
Known	Long	new	skips
Known	Long	follow-up	skips
Known	Short	new	reads
Known	Short	follow-up	reads
Unknown	Long	new	skips
Unknown	Long	follow-up	skips
Unknown	Short	new	reads
Unknown	Short	follow-up	skips

K Nearest Neighbours

KNN

- Training Data: $\{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
- KNN will make a **prediction** for a new object x_{new} based on the **closest** k examples from the training data
  - When $k = 1$, KNN predicts for x_{new} the label of the closest point in the training set
  - When $k > 1$, KNN uses a **consensus mechanism** to reach an agreement on some prediction
    - This is most commonly done through **majority voting**, if most points nearby meet one label - use that labael
- KNN **requires normalisation**
  - Particularly true when features of what we are testing have **very different scales** of how important they are

Generalisation

- We want the following things to be true in classification:
  - Small changes in training data should have a minimal effect on predictions
  - Classifies should work well for any test, not just one
  - Performance, as measured by kmP should remain consistent between training and testing
- KNN is generally a stable predictor
  - When k is not very small, the **voting mechanisms** keep it protected from being affected by changes in data
  - It's **non-parametric** nature helps too
    - It makes no assumptions about the functional form of the underlying data

Bagging and Random Forests

What is Bagging (Bootstrap Aggregating)

- Generate K distinct datasets by sampling **with replacement** from the dataset:
  - We randomly pick N examples from the training set, but each of those picks might be an example we picked before.

$$\mathcal{D} = \{(x_1, y_1), \dots (x_n, y_n)\}$$

- Each new dataset $\mathcal{D}_i$ is the same size as the training set: $|\mathcal{D}| = |\mathcal{D}_i| = n, \quad \forall i \in [1, 2, \dots, K]$ as it can have duplicates
- **Aggregation:** Combine the predictions of the models:
  - **Regression:** Average the results: $h_{bagging}(x) = \frac{1}{K} \sum_{i=1}^{K} h_i(x)$
  - **Classification:** Take a **majority vote** among the K classifiers.
- This is powerful for **reducing variance** on the classifiers h_i.
- This is problematic for standard **decision trees**:
  - If certain features lead to a large reduction of **information gain**, most trees h_i will greedily pick those features as the root.
  - Consequently, the trees will be highly **correlated** (look the same), limiting the variance reduction benefit.

Random Forests

- Improvement over bagging that **decorrelates** the trees.
  - Without randomness, bagging leads to **attributes** with very high **information gain** to be the root on every tree - which makes them less varied and therefore the aggregation less powerfuls
- Adds randomness in two places:
  - **Row Sampling:** Bagging (bootstrap samples of data).
  - **Feature Sampling:** At each split, consider only a random subset m of the total features p (typically $m \approx \sqrt{p}$).
- By forcing trees to use different features, the average of the trees becomes more robust.
- Random forests provide very good stability - even without pruning trees
- It is computationally efficient and **easily parallelisable**

Cross Validation

Cross-Validation

- CV is a method of using a training set for **training and validation**
  - Divid training set into K **non-overlapping** sets
  - Set aside 1 fold for validation, and train your classifier using other $K-1$ folds.
  - Repeat until every fold was used as validation set once
  - Report the average performance over the validation sets
- Average performance reported should induce **low bias** with respect to data

Comments on CV

- Using CV supposs two layers of model evaluation
  - Eval of model for **model selection** - using **only the training set**
  - Eval of the **selected** model on the test set
- Values of K are typically between 5 and 10
- CV is used in practive for **parameter tuning**
- When $K = n$, where $n = |\mathcal{D}|$, we can have an extreme case of CV:
  - Leave-One-Out cross-validation
  - It is very powerful but is clearly computationally exausting

Neural Networks

Josh Wilcox

November 25, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

Neural Networks

Tools So Far

- Baseline classifiers
  - KNNs
  - Decision Trees
- Ensemble Classifiers
  - Bagging
  - Random Forests
- Model Selection
  - KFold Cross-Validations

Parametric Classifiers

- **In simple terms:** Think of it like adjusting the knobs on a radio to get the best signal
  - The "parameters" (θ) are the knobs we can turn
  - We try different knob settings and measure how wrong our predictions are
  - The goal is to find the perfect knob settings that minimize our mistakes
- We look for optimal paramaters

$$\mathcal{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_\theta(\hat{y}^{(i)}), y^{(i)}$$

This calculates the average error across all our training examples

  - $\mathcal{J}(\theta)$ – The **cost function**: total score of how bad our predictions are
  - θ – The **parameters**: the adjustable settings in our model
  - N – The **number of training examples**: how many data points we have
  - $\mathcal{L}_\theta$ – The **loss function**: measures how wrong one single prediction is
  - $\hat{y}^{(i)}$ – Our **prediction** for example i
  - $y^{(i)}$ – The **actual correct answer** for example i

$$\theta^* = arg\min_\theta \mathcal{J}(\theta)$$

This finds the parameter values that give us the smallest error

  - θ^* – The **optimal parameters**: the best possible knob settings
  - $\arg\min$ – "**argument that minimizes**": which input value gives the smallest output

Perceptron

What is a Perceptron?

- **In simple terms:** A perceptron is like a simple decision-maker that draws a line to separate two groups
  - Imagine sorting emails into "spam" vs "not spam" by drawing a dividing line
  - It learns from its mistakes and adjusts the line each time it gets something wrong
  - Eventually, the line is positioned to correctly separate the two groups

Rules of the perceptron

- First Run: Randomize weights *(start with a random dividing line)*
- For each training sample (x, y):

$$\hat{y} = sign(w^\top x + b)$$

  Make a prediction: which side of the line is this point on?
  - x – The **input features**: the data point we're classifying (e.g., email characteristics)
  - y – The **true label**: the correct answer (+1 or -1, like "spam" or "not spam")
  - $\hat{y}$ – Our **prediction**: what we think the answer is
  - w – The **weights**: controls the angle/direction of our dividing line
  - b – The **bias**: shifts the line left or right (like the y-intercept)
  - $w^\top x$ – **Dot product**: combines the weights and input to get a score
  - $sign()$ – Converts the score to either +1 or -1 (forces a yes/no decision)
- If $\hat{y} \neq y$: *(if we predicted wrong, adjust the line)*
  - $w \leftarrow w + \eta y x$ *(adjust the angle of the line)*
  - $b \leftarrow b + \eta y$ *(shift the line left or right)*
  - η – The **learning rate**: how big of a step to take when correcting (like 0.01 = small careful steps)
  - $\leftarrow$ – means "update to" or "becomes"

$$sign(z) = \begin{cases} +1 & \text{if } z > 0 \\ -1 & \text{otherwise} \end{cases}$$

  The sign function: positive numbers become +1, negative numbers become -1

Properties

- Perceptrons are **purely linear**, only works for data that is **linearly separable** with one single line
- Real world data is often non-linear

Dealing with Non-Linear Data

The Problem

- Perceptrons can only draw straight lines, but real-world data rarely sits in neat straight-line patterns

The Solution: Non-Linear Transformations

- **In simple terms:** We ”warp” or ”reshape” the data space so that curved patterns become straight
- **How it's done:**
  - Apply mathematical functions to transform the input features
  - Common transformations: x^2 (squaring), $\sin(x)$, e^x, or combinations
  - Example: data in a circle pattern (x, y) can be transformed using $x^2 + y^2$
  - In the new space, previously curved boundaries become linear
- **Why we do this:**
  - Allows simple linear classifiers to solve complex problems
  - Real-world patterns (images, speech, text) are inherently non-linear
  - Makes previously impossible classification tasks solvable

Connection to Biological Neurons

- **Biological neurons:** Don't just add up signals – they transform them in complex ways
  - Brain neurons use chemical and electrical processes to create non-linear responses
  - They ”fire” only when stimulation exceeds a threshold (non-linear activation)
  - This allows brains to process complex patterns like faces, emotions, language

Biological Neurons

Biology

- Receives multiple stimuli, transmits signals to other neurons
- Perform highly complex non-linear transformations of their input stimuli
- **How biological neurons work:**
  - Dendrites receive electrical signals from other neurons
  - Signals are summed up in the cell body
  - If the total exceeds a threshold, the neuron "fires" (sends a signal)
  - The signal is transmitted through the axon to other neurons
  - This firing mechanism is inherently non-linear (all-or-nothing response)

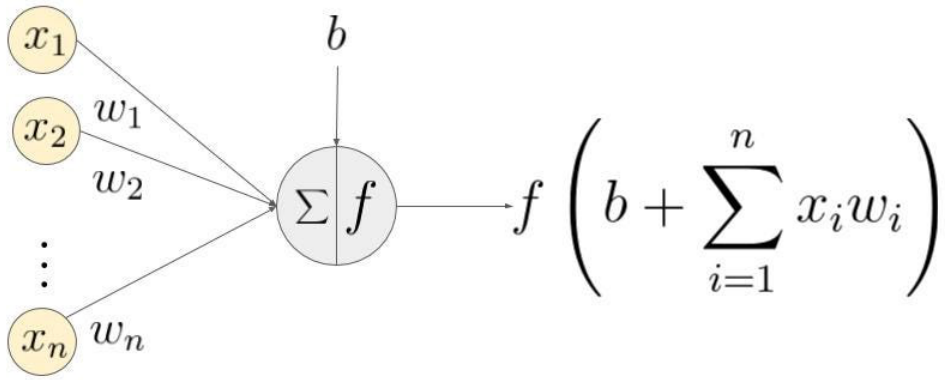
Artificial Neurons

- Current artificial neurons are highly abstracted versions of the brain's neural activity
- **The mathematical model:**

$$f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

This mimics the biological process: sum inputs, then apply non-linear transformation

  - x_i – The **inputs**: like signals from dendrites (input features $x_1, x_2, ..., x_n$)
  - w_i – The **weights**: strength of each connection (like synapse strength)
  - b – The **bias**: the threshold needed to activate (like firing threshold)
  - $\sum_{i=1}^{n} x_i w_i$ – **Weighted sum**: combines all inputs (like the cell body summing signals)
  - $f()$ – The **activation function**: creates non-linear response (like the firing mechanism)
- **Common activation functions:**
  - Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$ (smooth S-curve, outputs between 0 and 1)
  - ReLU: $f(z) = \max(0, z)$ (turns off for negative, linear for positive)
  - Tanh: $f(z) = \tanh(z)$ (S-curve, outputs between -1 and 1)
  - These create the non-linearity needed to solve complex problems



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

ReLu Function

What is ReLU?

- **ReLU** = Rectified Linear Unit
- **In simple terms:** A very simple on/off switch for neurons
  - If the input is negative, output zero (neuron is "off")
  - If the input is positive, pass it through unchanged (neuron is "on")
  - Like a one-way valve: blocks negative signals, allows positive ones

The Mathematical Definition

- Formula:

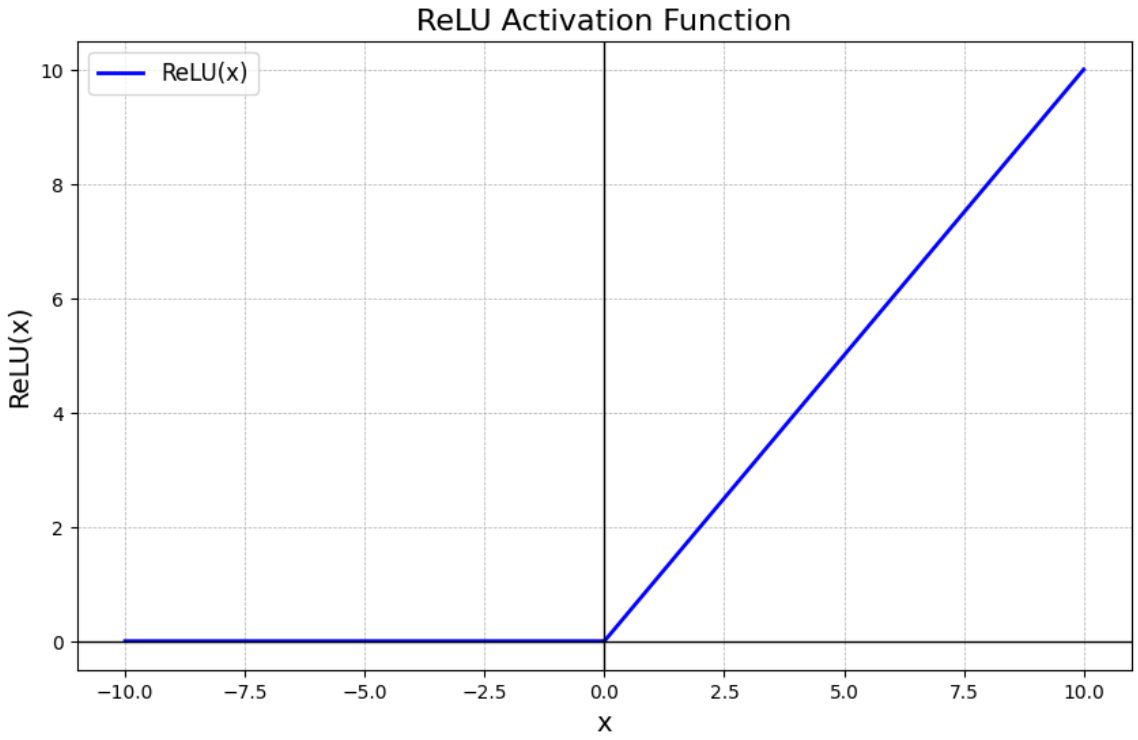
$$f(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Take the maximum of 0 and the input value

  - z – The input to the activation function (usually $b + \sum x_i w_i$)
  - $\max(0, z)$ – Chooses the larger value between 0 and z
  - Result: negative inputs become 0, positive inputs stay the same

Why ReLU is Important for Neural Networks

- **Provides non-linearity:** Without it, stacking layers would just be one big linear function
- **Computationally simple:** Just comparing with zero (faster than sigmoid or tanh)
- **Solves the "dying gradient" problem:** Helps deep networks learn better
- **Sparse activation:** Many neurons output zero, making the network more efficient
- **Most popular:** Used in the majority of modern deep learning models (CNNs, transformers, etc.)



Graph shows: flat at 0 for negative inputs, diagonal line for positive inputs

Multi-Layer Perceptron

What is a Layer?

- **In simple terms:** A layer is a group of artificial neurons working in parallel
  - Think of it like a team where each member looks at the same information but focuses on different aspects
  - Each neuron in a layer performs its own calculation: $f(b + \sum x_i w_i)$
  - All neurons in a layer receive the same inputs but have different weights
  - The outputs from one layer become the inputs to the next layer

---

Structure of Multi-Layer Perceptron

- **Input Layer:**
  - Takes in your raw data (like pixel values, measurements, etc.)
  - Just passes the data forward – no computation happens here
  - Number of neurons = number of input features
- **Hidden Layer(s):** (Two or more)
  - This is where the ”magic” happens – neurons learn patterns
  - Each neuron is a complete artificial neuron with its own weights and bias
  - First hidden layer learns simple features (edges, colors)
  - Deeper layers learn complex combinations (shapes, objects)
  - Called ”hidden” because we don’t directly see what they learn
- **Output/Decision Layer:**
  - Produces the final prediction or classification
  - Number of neurons = number of classes (e.g., 2 for spam/not spam)
  - Uses outputs from hidden layers to make the final decision

---

Why Multiple Layers Matter

- **Single perceptron:** Can only draw one straight line (very limited)
- **Multiple layers:** Can learn incredibly complex patterns
  - Each layer applies non-linear transformations
  - Stacking these transformations lets the network ”bend” the decision boundary
  - This is how neural networks recognize faces, understand speech, play chess
- **Deep learning** = many hidden layers (hence ”deep” neural networks)

Training MLPs

The Training Process

- **In simple terms:** Training is like teaching through trial and error
  - Show the network examples, let it guess, tell it how wrong it was
  - The network adjusts its weights to reduce future mistakes
  - Repeat thousands of times until it gets really good at predictions

Step 1: Initialize

- **Randomize weights and biases**
  - Start with random small values (like $w \sim \mathcal{N}(0, 0.01)$)
  - Can't start at zero – all neurons would learn the same thing!
  - This randomness ensures each neuron learns different patterns

Step 2: Forward Pass (For each training example)

- **Network applies operations in hidden layers:**
  - Each layer: compute weighted sum, then apply activation function
  - Layer l: $h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)})$
  - Data flows forward: input $\rightarrow$ hidden layers $\rightarrow$ output
  - $h^{(l)}$ – **Hidden layer activations**: outputs from layer l
  - $W^{(l)}$ – **Weight matrix** for layer l: connections between layers
  - $b^{(l)}$ – **Bias vector** for layer l: thresholds for each neuron
  - $f()$ – **Activation function**: usually ReLU for hidden layers

Step 3: Make Prediction

- **Final output layer prediction:**

$$\hat{y} = \text{softmax}(W^{(L)}h^{(L-1)} + b^{(L)})$$

Convert the final layer's scores into probabilities for each class

  - $\hat{y}$ – **Predicted probabilities**: one for each class (sums to 1)
  - L – **Last layer number**: the output layer
  - $h^{(L-1)}$ – **Last hidden layer output**: input to final layer
  - softmax – Converts scores to probabilities: $\frac{e^{z_i}}{\sum_j e^{z_j}}$

Training MLPs (continued)

Step 4: Compute the Loss

- **Quantify how wrong the prediction was:**

$$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i)$$

Cross-entropy loss: measures difference between predicted and true probabilities

  - $\mathcal{L}$ – **Loss value**: single number measuring wrongness (lower = better)
  - y_i – **True label** (one-hot encoded): 1 for correct class, 0 for others
  - $\hat{y}_i$ – **Predicted probability** for class i
  - If prediction is confident and correct: loss is low
  - If prediction is confident but wrong: loss is high (big penalty!)

Step 5: Backpropagation

- **Compute gradient of loss with respect to ALL parameters:**

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

How much does each weight/bias contribute to the error?

- **Why partial derivatives?**
  - The loss depends on THOUSANDS of parameters simultaneously
  - **Partial derivative** = how loss changes when we adjust ONE parameter, keeping all others fixed
  - Like checking "if I turn just this knob, does the error go up or down?"
  - Tells us the direction and magnitude to adjust each weight
- **How backpropagation works:**
  - Uses the **chain rule** from calculus to propagate gradients backward
  - Start at output: compute $\frac{\partial \mathcal{L}}{\partial W^{(L)}}$ (easy – close to loss)
  - Move backward: use chain rule to get $\frac{\partial \mathcal{L}}{\partial W^{(L-1)}}$, then $W^{(L-2)}$, etc.
  - Each layer passes gradient information to the previous layer
  - Efficient: computes ALL gradients in one backward pass!
- **Update weights using gradient descent:**

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

Move parameters in the opposite direction of the gradient (downhill)

  - η – **Learning rate**: step size (e.g., 0.001 = small careful steps)
  - Negative sign: gradient points uphill, we want to go downhill (minimize loss)
  - Each parameter gets nudged slightly toward lower error

Step 6: Repeat

- Go back to Step 2 with next training example
- Continue for many epochs (passes through all training data)
- Network gradually learns better and better representations

Convulutional Neural Networks

What are Convolutional Neural Nets

- Architecture designed for **structured data**
- Replace dense layers with **convolutional filters**
  - These detect local patterns
  - Filters are then applied to full images through weights sharing
- Captures **spatial hierarchy**
  - Early layers in the net learn simple features
  - Deeper layers learn complex and abstract objects
- They implement **pooling layers**
  - These summarize very important features

Recurrent Neural Networks

- Used for **sequential data**
- Maintain a **hidden memory** - designed for long-running training sequences
- Hidden memory from one step is fed back into the next step along with the new input

Natural Language Processing

Josh Wilcox

November 27, 2025

Thank you for taking a look at my notes!

Please give my sites a visit and interact/star things if you find my notes useful

Website: josh.software

Instagram: joshwilcox.md

LinkedIn: linkedin.com/in/josh-wilcox-swe

GitHub: github.com/Joshua-Wilcox

Your engagement and feedback help me a lot

Table of Contents

NLP

What is NLP

- Broad term that encompasses all things that handle **computers** and **language**
  - Both understanding and generating

Language Modelling

- Language modelling **assigns probability to text**
- For example in a **finite vocabulary** of words $V = \{\text{these, are, example, words}\}$
  - We can construct an infinite set of strings V^*
  - Then we can estimate a probability distribution:

$$V^* \to \mathbb{R} \text{ s.t. } \sum_{s \in V^*} p(s) = 1;\ \forall s \in V^*,\ p(s) > 0$$

Unigram Language Model

- This is a language model where **sentences are finite** with a "STOP" token
- Each word is generated with **indipendantly and identically distributed** with probability $q(x_i)$

$$p(s = x_1, x_2 \ldots, x_n) = \prod_{i=1}^{n} q(x_i)$$

$$V^* \to \mathbb{R} \text{ s.t. } \sum_{x_i \in V^*} p(x_i) = 1;\ \forall s \in V^*,\ p(x_i) > 0$$

- This is **generative** as you can keep choosing **the most probable** word until "STOP" is chosen

Bigram Language Model

- Similar to **unigram**, but each word **depends only on its predecessor**

$$p(s = x_1, x_2 \ldots, x_n) = \prod_{i=1}^{n} q(x_i | x_{i-1})$$

$$V^* \to \mathbb{R} \text{ s.t. } \sum_{x_i \in V^*} p(x_i) = 1;\ \forall s \in V^*,\ p(x_i) > 0$$

N-Gram Language Model

- Each word depends on $N - 1$ predecessors

$$p(s = x_1, x_2 \ldots, x_n) = \prod_{i=1}^{n} q(x_i | x_{i-1}, \ldots, x_{i-N+1})$$

$$V^* \to \mathbb{R} \text{ s.t. } \sum_{x_i \in V^*} p(x_i) = 1;\ \forall s \in V^*,\ p(x_i) > 0$$

Tri-Gram example

Trigram (3-Gram) Example

- For a **trigram model**, each word depends on the **previous two words**
- We use **START** tokens to handle the beginning of a sentence
- Consider the sentence: `"the dog barks"`

$$\begin{aligned} p(\text{the dog barks STOP}) = & \, q(\text{the} \mid \text{START}, \text{START}) \\ & \times q(\text{dog} \mid \text{START}, \text{the}) \\ & \times q(\text{barks} \mid \text{the}, \text{dog}) \\ & \times q(\text{STOP} \mid \text{dog}, \text{barks}) \end{aligned}$$

- Each conditional probability $q(w_i \mid w_{i-2}, w_{i-1})$ is estimated from corpus counts:

$$q(w_i \mid w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})}$$

Evaluating LM Performance

Criteria

- **Likelihood** - Probability of training data under the model:

$$p_{\mathcal{M}}(\mathcal{D}')$$

- **Log Likelihood** - Log of probability of data w.r.t model

$$log(p_{\mathcal{M}}(\mathcal{D}'))$$

- **Perplexity** - Inverse prob of data, normalised by number f words

$$PP_M(D') := P_M(D')^{-k}$$

Goal

- We want to find the model that maximizes **likelihood** on some training set $\mathcal{D}$

$$\hat{M} = \arg\max_{\mathcal{M}} p_{\mathcal{M}}(\mathcal{D})$$

Equivalence of Metrics

- Optimizing all three metrics (**likelihood**, **log likelihood**, **perplexity**) are **equivalent**

Likelihood ⇔ Negative Log Likelihood:

$$\begin{aligned}\hat{M} &= \arg\max_{\mathcal{M}} p_{\mathcal{M}}(\mathcal{D}) \\ &= \arg\max_{\mathcal{M}} \log p_{\mathcal{M}}(\mathcal{D}) \quad \text{(log is monotonic increasing)} \\ &= \arg\min_{\mathcal{M}} -\log p_{\mathcal{M}}(\mathcal{D})\end{aligned}$$

⇒ **Maximizing likelihood** is equivalent to **minimizing negative log likelihood**

Likelihood ⇔ Perplexity:

$$\begin{aligned}\hat{M} &= \arg\max_{\mathcal{M}} p_{\mathcal{M}}(\mathcal{D}) \\ &= \arg\min_{\mathcal{M}} p_{\mathcal{M}}(\mathcal{D})^{-1} \quad \text{(inverse is monotonic decreasing)} \\ &= \arg\min_{\mathcal{M}} \sqrt[k]{p_{\mathcal{M}}(\mathcal{D})^{-1}} \quad (k\text{-th root is monotonic increasing}) \\ &= \arg\min_{\mathcal{M}} p_{\mathcal{M}}(\mathcal{D})^{-\frac{1}{k}} = \arg\min_{\mathcal{M}} PP_{\mathcal{M}}(\mathcal{D})\end{aligned}$$

⇒ **Maximizing likelihood** is equivalent to **minimizing perplexity**

Further Explanation:

- **Why are these equivalent?** All three metrics are different mathematical ways to measure how well a language model predicts the data. They are monotonic transformations of each other, so optimizing one optimizes the others.
- **Likelihood** is the direct probability assigned to the data by the model. Higher likelihood means the model predicts the data better.
- **Log likelihood** is used because probabilities can be very small, and logs turn products into sums, making optimization easier and numerically stable. Since the log function is monotonic, maximizing log likelihood is the same as maximizing likelihood.
- **Negative log likelihood** is often minimized in practice (e.g., as a loss function in machine learning) because most optimizers are designed to minimize functions.
- **Perplexity** is a normalized, interpretable version of likelihood. It measures how "surprised" the model is by the data: lower perplexity means better predictions. Minimizing perplexity is mathematically equivalent to maximizing likelihood.

LLMs

What are LLMs

- Built on **Transformer** architecture trained on internet-scale data
- Most are proprietary models, but open source models are catching up
- They cost many millions of dollars to trains, and require 1000s of expensive GPU hardware

Building an LLM

- **Get internet scale data**
  - Uses a mixture of low-quality (high volume) and high quality (low volume) data
- **Implement Tokenization**
  - Transforms text to a long list of integers
  - Many words can map to one token
  - Sequences of characters commonly found next to each other may be grouped
- **Pre-Training**
  - Train the transformer on all the data we give it
  - Outcome is just a completion robot, not trained for any specific task in mind
- **Supervised Fine tuning**
-
  - Helps making it more an assistant
  - Fine tunes to a more human, high quality data set
  - Makes it actually able to complete a task like text classification etc.

RLHF

- Comprises of **reward modelling** and **reinforcement learning step**
- **Reward Modelling**: Train a separate model to score outputs based on human preferences
  - Humans rank multiple generated responses for quality, helpfulness, and safety
  - The reward model learns to predict these rankings
- **Reinforcement Learning Step**: Use PPO (Proximal Policy Optimization) to fine-tune the LLM
  - The LLM generates responses, reward model scores them, and policy is updated to maximize rewards
  - Helps align the model with human values, reducing harmful outputs

DPO

- **Direct Preference Optimization (DPO)** is a recent alternative to RLHF for aligning language models with human preferences
- Instead of using reinforcement learning, DPO directly optimizes the model to prefer outputs that humans rate higher
  - Collect pairs of responses: one preferred by humans, one less preferred
  - Train the model to assign higher probability to the preferred response over the less preferred one
  - Uses a simple loss function based on the difference in log probabilities between preferred and non-preferred outputs
- **Advantages:** DPO is simpler, more stable, and avoids the complexity of reinforcement learning algorithms like PPO

The Transformer Architecture

Overall Concept

- Introduced in "Attention is All You Need" (Vaswani et al., 2017)
- Moves away from Recurrent Neural Networks (RNNs) entirely
- Relies solely on **Attention mechanisms** to draw global dependencies between input and output
- **Parallelizable**: Unlike RNNs, the entire sequence can be processed at once during training

Positional Encoding

- Since there is no recurrence, the model has no inherent sense of **order** or sequence
- We inject information about the relative or absolute position of the tokens in the sequence
- Added directly to the input embeddings:

$$\text{Input} = \text{Embedding} + \text{PositionalEncoding}$$

Self-Attention

- **Goal**: Enrich the embedding vector with context
- Allows the model to look at other words in the sentence to better understand the current word
- Example: In "The animal didn't cross the street because **it** was too tired", self-attention associates "**it**" strongly with "**animal**"
- Computes relationships/dependencies between all tokens in the input simultaneously

Layer Normalization & Residuals

- Deep networks are hard to train; these components stabilize and accelerate convergence
- **Residual Connections**: The input of a layer is added to its output:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

- Allows gradients to flow through the network more easily

The Encoder (Understanding)

- Processes the **entire input sequence** at once
- Maps input symbols (tokens) into a continuous abstract representation
- Consists of a stack of layers (Self-Attention + Feed Forward Neural Networks)
- **Output**: A rich contextual matrix representing the meaning of the input, which is passed to the Decoder

The Decoder (Generation)

How it Predicts the Next Word

- The Decoder is **autoregressive**: it generates one token at a time
- Uses two sources of information:
  - The **Representation** created by the Encoder (Context)
  - The **Previously Generated Tokens** (History)

Masked Self-Attention

- During training, the model has access to the full target sentence
- We must prevent the model from "cheating" by seeing future words
- **Masking**: Future positions are set to $-\infty$ in the attention scores so they result in 0 probability
- Ensures predictions are made only using **known information** (past tokens)

Encoder-Decoder Attention

- This is the bridge between input and output
- Helps **align** the input sequence with the output generation
- The Decoder looks back at the Encoder's representation to decide which parts of the input are relevant for the *current* word it is generating